ОБ ОБЩИХ ПОДХОДАХ К РАЗРАБОТКЕ ГРАФИЧЕСКОГО ФРЕМВОРКА ДЛЯ МОБИЛЬНОЙ ПЛАТФОРМЫ

Г.А. Ломакин, Л.В. Рудикова
Кафедра программного обеспечения интеллектуальных и компьютерных систем
Учреждение образования «Гродненский государственный университет имени Янки Купалы»
Гродно, Беларусь
E-mail: {rudikowa, spellbound.fpmi}@gmail.com

В предлагаемой статье предложен подход к построению графического фреймворка для мобильной платформы на основе ОС Android. Рассматриваются основные аспекты реализации программного обеспечения, связанного визуализацией трехмерных объектов. Описывается общая архитектура разрабатываемого фреймворка.

Введение

В настоящее время существуют стандартные подходы к разработке крупных графических приложений. Крупные компании самостоятельно разрабатывают необходимые решения, а для всех остальных на рынке программного обеспечения предлагаются различные универсальные решения для использования, например, Unity 3D. Предлагается разработка представляет собой фреймворк на OpenGL с поддержкой платформы Android, который предоставляет открытый код и возможность легкого расширения. Кроме того, предлагаемый функционал является, в достаточной мере, гибким и используемым. Абстрактная модель строится таким образом, чтобы все наиболее используемые возможности были легко доступны пользователю.

I. Основные подходы к разработке

Одной из основных проблем при разработке подобных решений является реализация решения, отвечающего за позиционирование объектов в пространстве. В качестве такого решения были выбраны и доработаны кватернионы и системы кватернионов.

Существует два основных метода задания поворота 3Dобъекта в пространстве: углы Эйлера и кватернионы. Для начала рассмотрим углы Эйлера. Главная идея заключается в повороте объект вокруг заданной оси на определенный угол. Таким образом, для задания поворота нам необходимо выбрать вектор и угол. Чтобы описать любое положение объекта в пространстве, необходимо выполнить несколько подобных операций. Для 3D-мерного пространства можно выбрать три вектора – X, Y и Z, после этого провести повороты относительно этих заданных векторов. Однако тут возникает один аспект – конечный результат зависит от порядка вращения. Таким образом, если будем производить вращение в порядке XYZ, то не обязательно оно будет в точности совпадать с вращением XZY или ZXY. Получается, что необходимо вращать ось последующего вращения на тот же угол относительно предыдущей оси вращения.

Отметим, что кватернион также можно представит в виде матрицы. Так как OpenGL работает с матрицами, то перед их использованием необходимо конвертировать кватернион в матрицу. Таким образом, получаем следующий алгоритм – вся математика поворотов проходит в кватернионах, а затем конвертируется в матрицу и передается OpenGL.

На практике имеем следующее: для конкретного 3D-объекта хранится три скаляра – повороты относительно XYZ. Далее строятся три кватерниона – повороты относительно каждой из оси. После этого кватернионы перемножаются, и получается кватернион конечного поворота, из которого извлекается матрица. В дальнейшем на эту полученную матрицу и будет происходить умножение при формировании матрицы модели.

Положительные аспекты предлагаемого решения – математика кватернионов проще и более стабильная при реализации, чем углы Эйлера. Однако, это влечет немного большие затраты по производительности.

II. Алгоритм ShadowMapping

Наиболее интересное графическое решение было выбрано для генерации освещения в реальном времени. Алгоритм основан на базе классического Shadowmapping с некоторыми доработками для сглаживания теней. Данный алгоритм описывает отрисовку теней для сцены в реальном времени. Алгоритм состоит из двух этапов – генерация карты глубины из направленного источника света и отрисовки сцены от лица камеры с учетом карты затенения.

Первый этап. Сначала генерируется карта глубины из источника света. Если говорить об одном источнике освещения, то в тени будет все то, что «не видно» с позиции источника света с учетом его направления. Отсюда следует необходимость генерировать карту глубины из источника освещения. Таким образом, получаем данные о затенении для последующего рендерин-

Библиотека БГУИР

га. Для точечного источника света генерируется CubeMap из шести текстур глубины – верх-низ-право-влево-вперед-назад.

Второй этап. Зная карту глубин и расположение источника света, можно приступать к рендерингу. При отрисовке необходимо учитывать глубину, но, чтобы правильно ее извлечь, необходимо позицию конечного пикселя привести в систему координат источника освещения. Для этого необходимо определить и передать в шейдер матрицу MVP для источника освещения.

После применения матрицы к позиции пикселя получим 2D-координаты в системе освещения. Используя эти координаты, можно получить глубину затенения в этой точке и, если выбранный пиксель окажется с меньшей глубиной относительно источника света, чем сгенерированная ранее глубина в этой позиции, то пиксель будет освещен, иначе – затенен.

III. Основная архитектура фреймворка

Главная идея предлагаемого фреймворка – расширенные возможности по сборке контента. Изначально определяются несколько отдельных сущностей – Текстура, Шейдер, Меш, Логика обновления, Логика отрисовки позволяет комбинировать все эти сущности между собой и получать на выходе требуемый результат.

Архитектура фреймворка строится таким образом, что все эти сущности не связаны между собой. Основу разработки составляют несколько, так называемых, Store-хранилищ, в которых содержатся основные данные, заложенные в основу отрисовок. Итак, выделены три основных хранилища – Меши (3D-моди, Mesh), Текстуры и Шейдеры. При необходимости можно добавить также хранилище и для другого типа хранимых объектов. Все хранилища наследуются от базового хранилища, которое является абстрактным и типизируемым, которое также поддерживает целостность данных, защищено от OutOfMemory и других исключительных ситуаций. Рассмотрим указанные сущности.

Mesh. Включается в себя наборы данных о вершинах – текстурные координаты, вектора нормали для каждой вершины и индексы для отрисовки. Для Mesh реализован специальный класс MeshLoader, который загружает их из файла (формат MyGL, разработан специально для движка с целью минимизации данных в файлах описания Mesh), а также собирает Mesh из массивов текстурных координат нормалей и остальных данных. MeshLoader адаптирован под работу с нативными структурами данных, так как структуры данных в Java содержат избыточную реализацию и проблемы, связанные с производительность при их использовании.

Texture. Загружается из форматов png, jpg, gif. Важной особенностью является то, что размеры текстуры должны быть кратны степеням числа 2, так как не все графические адаптеры работают с текстурами других размеров.

Shaders. Класс-обертка над программами, выполняемыми напрямую на GPU. Создаются с использованием ShaderLoader, который компилирует шейдеры из файловой системы и связывает с программой для GPU. Shaders также содержит всю логику по инициализации конкретных шейдеров и предоставляет гибкие возможности для использования и масштабирования.

Логика обновления. Перед тем как отрисовывать объект, необходимо учесть прошедшее время и рассчитать его положение в пространстве, а также другие изменения, которые могли произойти за это время. Для разных объектов реализуется совершенно разная логика поведения. Однако можно выделить общие абстрактные зависимости. Например, объекты должны реагировать на столкновения с другими объектами. Для таких целей разработан CollisionController, который обрабатывает такие события.

Логика отрисовки. Разные объекты необходимо отрисовывать по-разному, учитывая тени, альфа-канал, материал, постобработку, а также их комбинации. В силу этого логика отрисовки вынесена как отдельная сущность, общие черты которой можно выявить и составить эффективную иерархию наследования.

Итак,разработанное ядро CoreX содержит в себе информацию о текущей загруженной сцене, а также хранилища для всего контента [1]. Сцена представляется собой набор объектов (комбинаций сущностей), которые «знают», как себя вести за все время своего жизненного цикла. Предлагаемая конструкция позволяет достаточно просто генерировать многообразие объектов, используя абстрактные типы поведения и отрисовки, связывая их с текстурами, мешами и произвольным шейдером для материала.

Заключение

Предлагаемый фреймворк разрабатывается как альтернатива существующим без использования единого кода для разных платформ. В структуре движка заложена архитектура и нейминги, которые позволяют перенести фреймворк на другую платформу или, даже, язык программирования с минимальными изменениями. Кроме этого, разрабатываемый продукт предполагается не слишком абстрактным, но позволяет добавлять мало используемые функции. В тоже время, предполагается возможность расширения предлагаемого продукта сторонними разработчиками и выпуск некоторых плагинов и сборок.

1. Ломакин, Г.А. Об одном подходе к построению графических приложений / Г.А. Ломакин // Наука-2011 : сб науч. ст. В 2ч. Ч. 2 / БГУ ; рекол.: В.И. Малюгин (отв. ред.) [и др.]. – Минск : БГУ, 2011. – С. 89-91.

Библиотека БГУИР